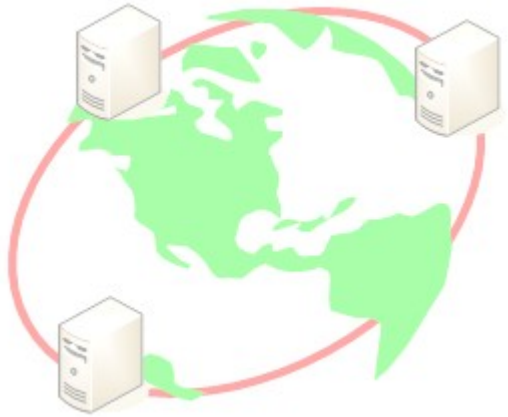
Sistemas Distribuídos
Message-Passing Interface

Universidade Federal do ABC

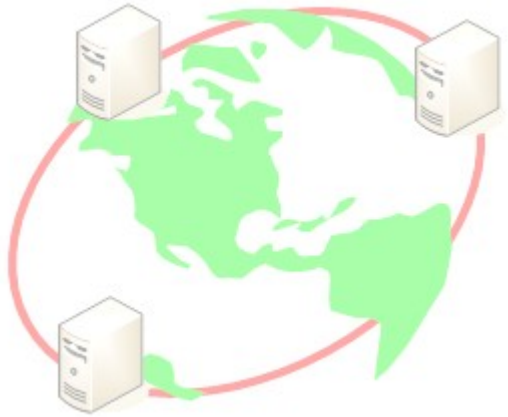
Turma:
Ciência da Computação

Prof. Dr. Francisco Isidro Massetto



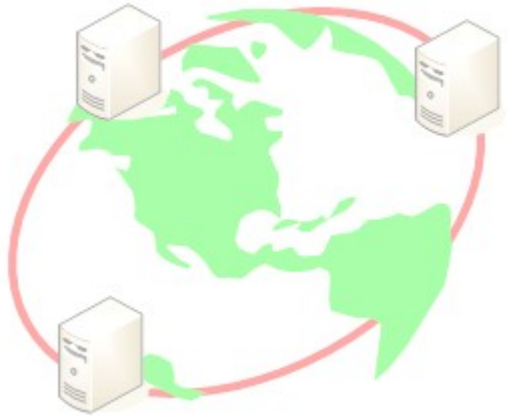
Cluster Computing

- Cluster
  - Aglomerado de máquinas (nós)
  - Visão de único sistema (single-image system)
  - Implementação de Sistema Distribuído
  - Utilização
    - Necessidade de alto poder computacional
      - Computação gráfica
      - Análise de elementos finitos
      - Sensoramento remoto
      - Exploração sísmica



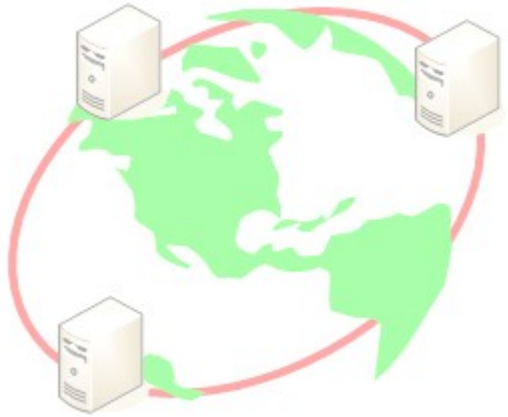
Cluster Computing

- Utilização
  - Necessidade de tolerância a falhas
    - Segurança de Dados
    - Internet
    - Disponibilidade
    - Descentralização de atividades

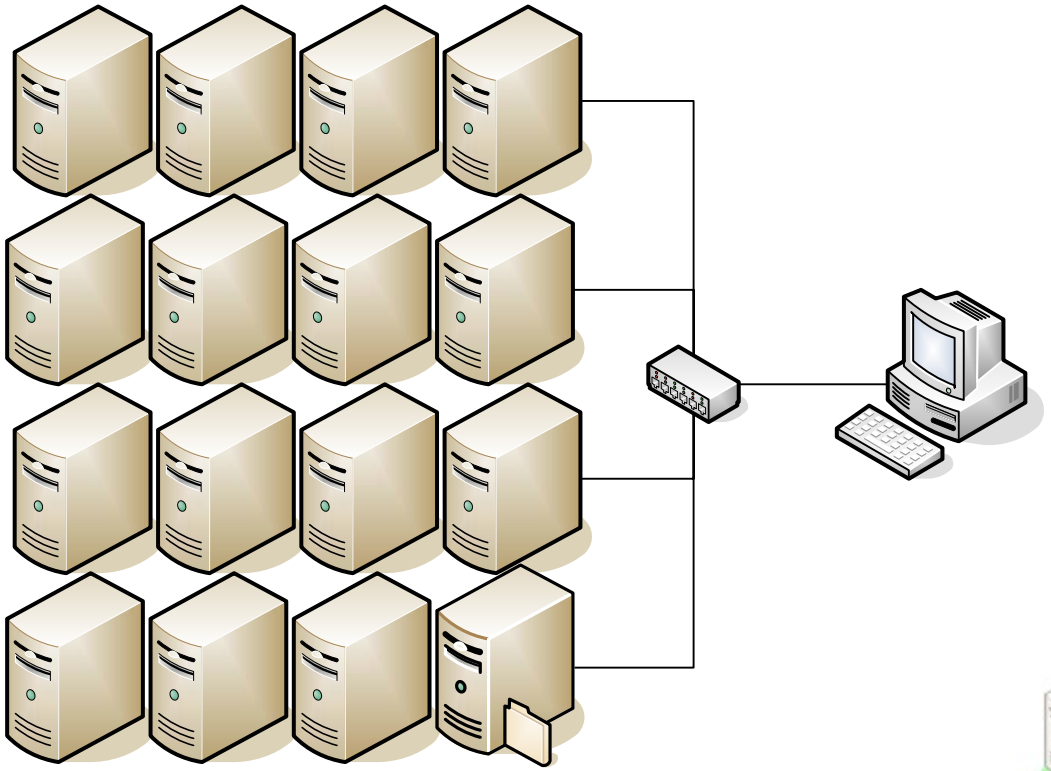


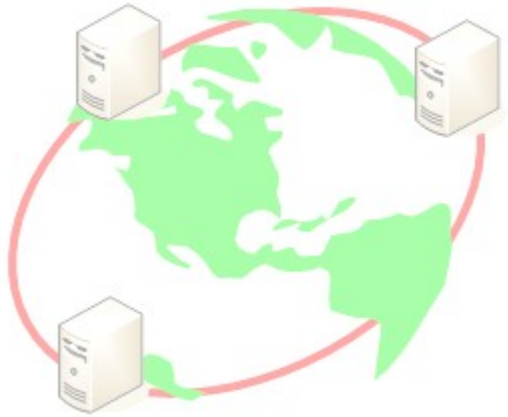
Desafio

- Programar Aplicações que demandem alto poder de processamento
- Paralelização dos dados
- Distribuição
- Ausência de memória global compartilhada



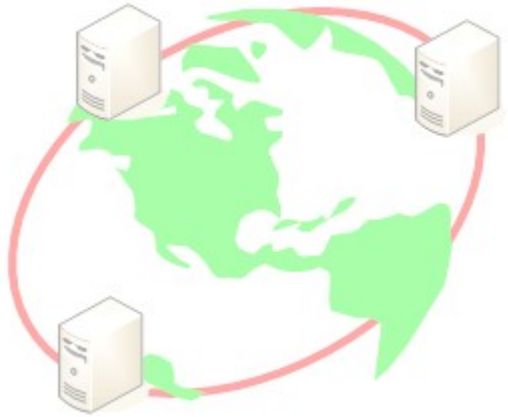
Arquitetura





MPI

- Message-Passing Interface
- Interface que implementa primitivas de comunicação ponto-a-ponto e coletivas
- Responsável por todo o processo de estabelecer comunicação
- Abstrai aspectos de protocolos de transporte

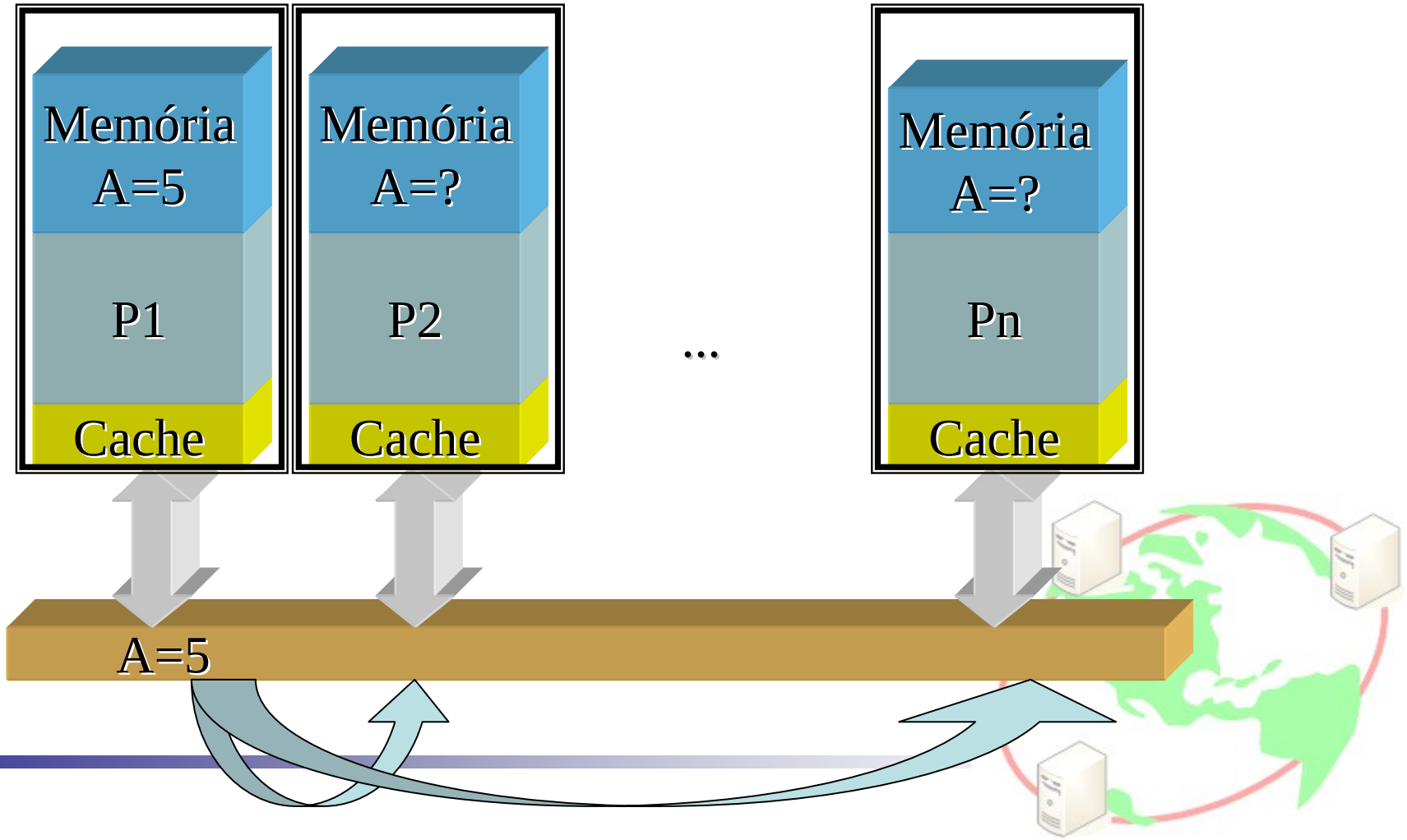


MPI

- Introduzido pelo MPI Forum em maio de 1994
- atualizado em junho de 1995.
- Incluiu a participação de vendedores de hardware, pesquisadores, acadêmicos, desenvolvedores de bibliotecas de software, e usuários
- "MPI: A Message-Passing Standard",
  - University of Tennesee,
  - Argonne National Laboratory.

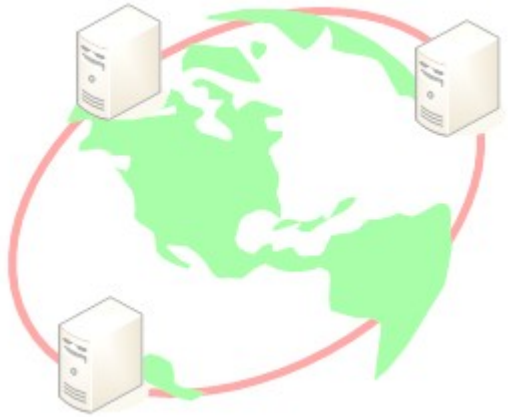
HPCLab

Arquiteturas Distribuídas



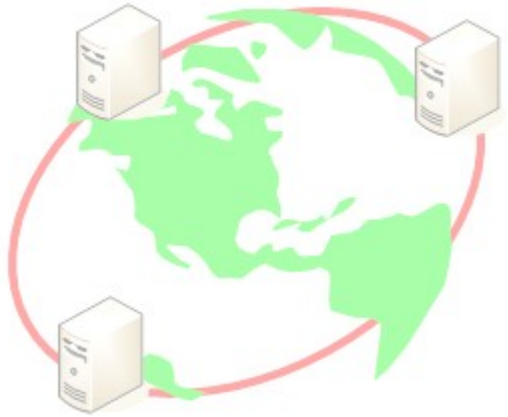
Identificação de Processos

- Cada processo é identificado por um “rank”
- Todos os processos são disparados (através de execução remota) nas máquinas determinadas
- Uma máquina pode executar vários processos (não necessariamente um em cada máquina)
- N processos → ranks de 0 a N-1



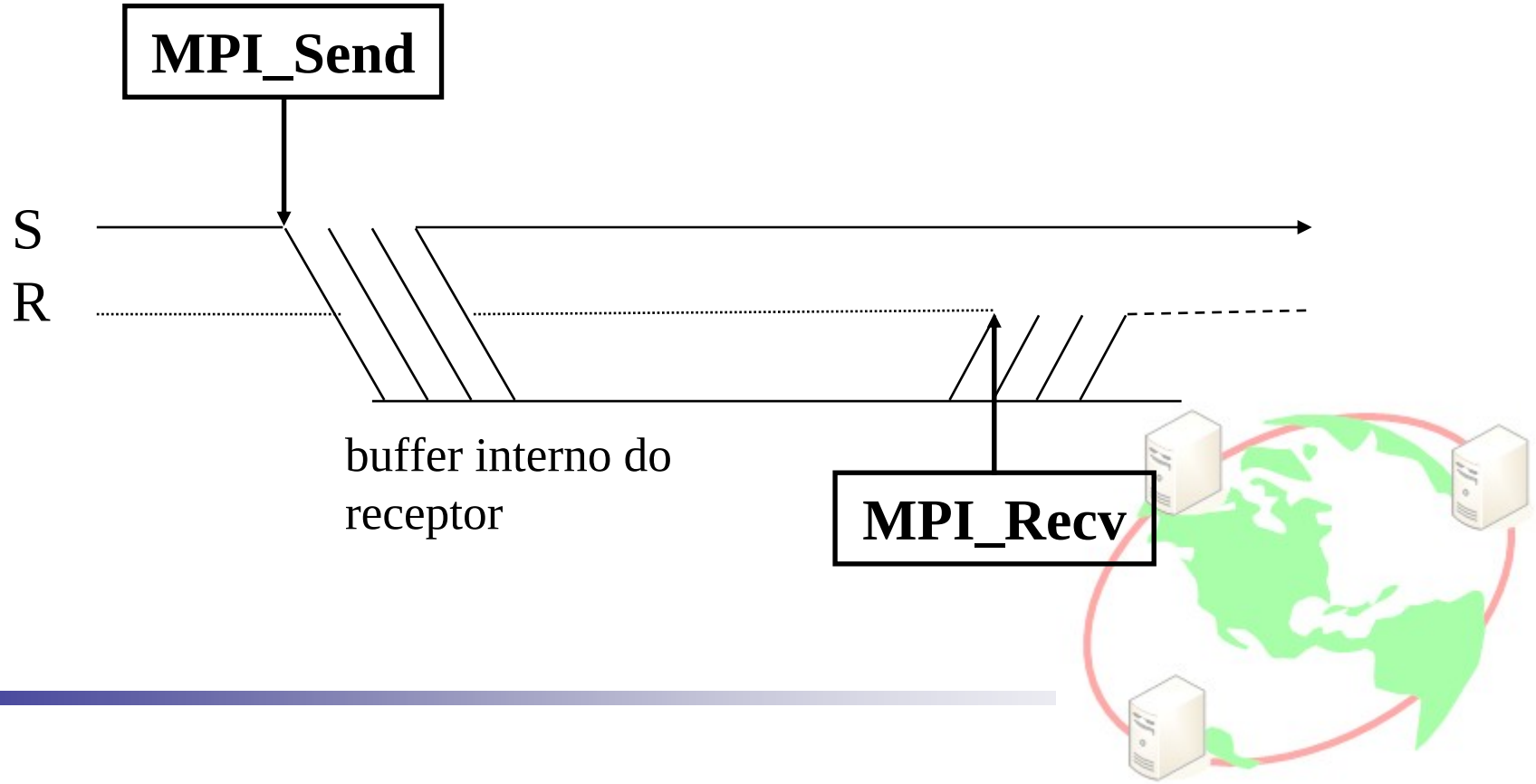
Tipos de Primitivas

- Mais de 125 primitivas ao todo
- MPI_Init
  - inicialização para o ambiente MPI
- MPI_Comm_size
  - retorna o número de processadores
- MPI_Comm_rank
  - retorna o "rank" (índice, identificador) do processador
- MPI_Send
  - envia uma mensagem
- MPI_Recv
  - recebe uma mensagem
- MPI_Finalize
  - sai do ambiente MPI



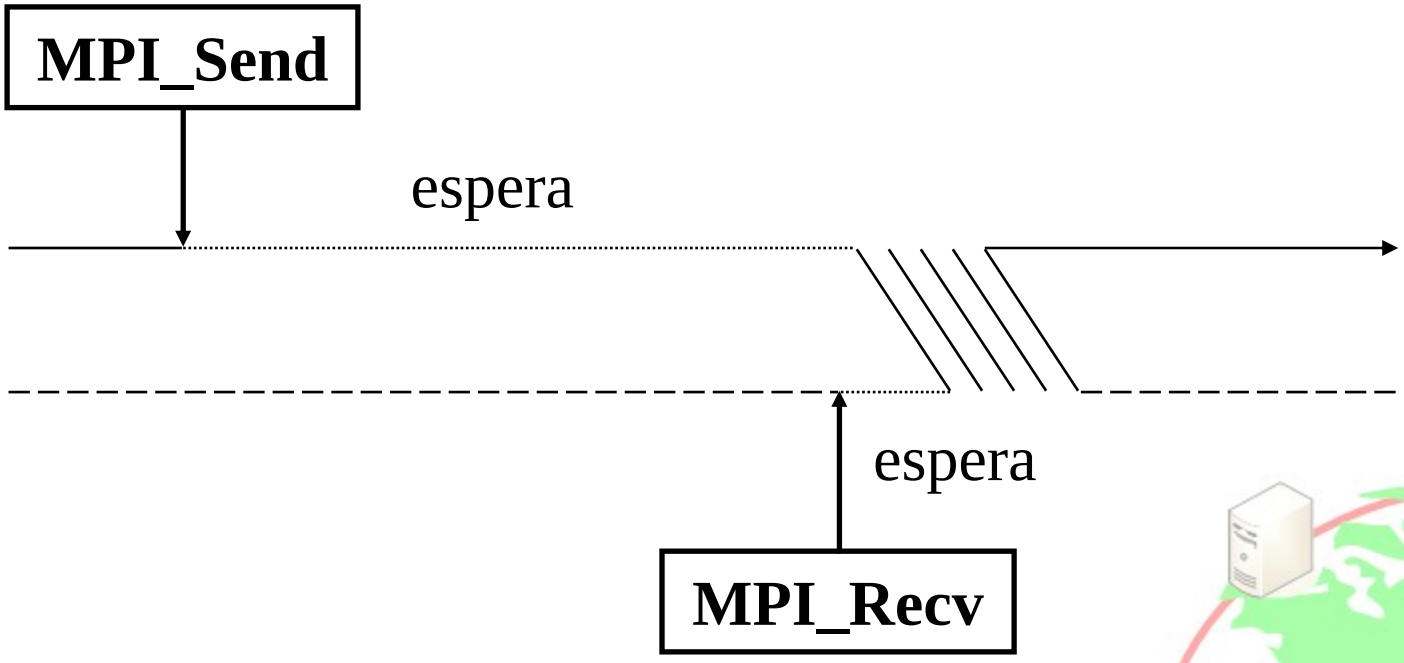
MPI

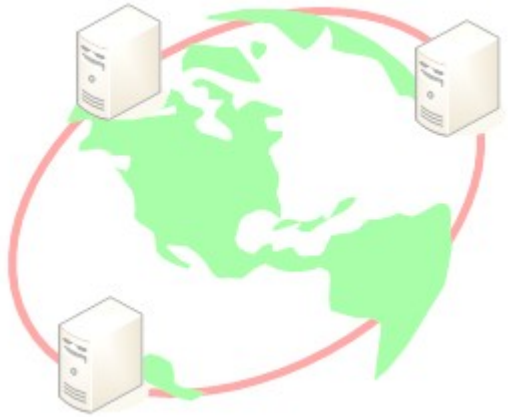
- Tamanho da mensagem < Limite



MPI

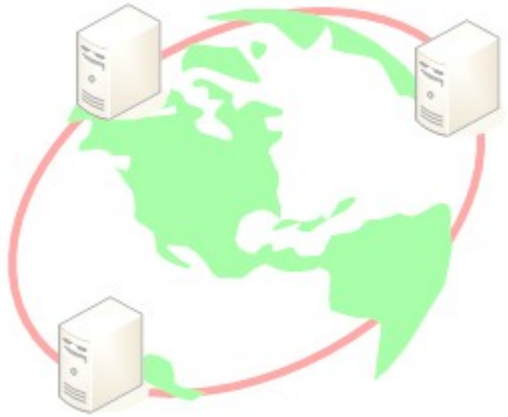
- Tamanho da Mensagem > Limite





Tipos Básicos de Dados MPI

- MPI_CHAR e MPI_UNSIGNED_CHAR
- MPI_SHORT e MPI_UNSIGNED_SHORT
- MPI_INT e MPI_UNSIGNED
- MPI_LONG e MPI_UNSIGNED_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE
- MPI_BYTE
- MPI_PACKED



HPCLab

Principais Primitivas MPI

- Iniciação e Finalização

Número de argumentos

Argumentos (vetor)

MPI_Init (int *argc, char ***argv)

MPI_Finalize (void)

Sem parâmetros

Exemplo:

```
void main(int argc;char *argv[])
{
   MPI_Init (&argc, &argv);
   ...
   MPI_Finalize ();
}
```

Principais Primitivas MPI

- Número de Processos

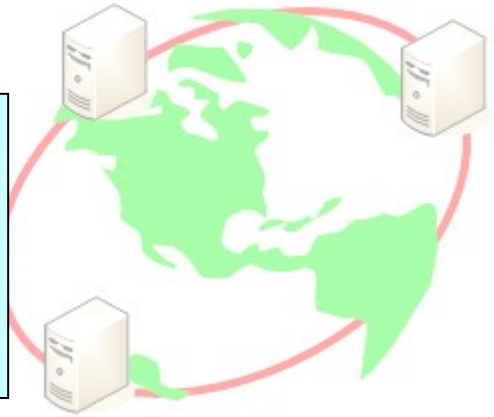
Comunicador MPI

```
MPI_Comm_size (MPI_Comm comm, int *size)
```

Variável de retorno

Exemplo:

```
int ntasks;
MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
```



Principais Primitivas MPI

- Identificação do Processo Vigente

Comunicador MPI

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Variável de retorno

Exemplo:

```
int rank;
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

Principais Primitivas MPI

- Envio de Mensagens

Endereço inicial dos dados que serão enviados

Número de elementos a serem enviados

Tipo do dado

```
MPI_Send (void *buf, int count, MPI_Datatype dtype,
          int dest, int tag, MPI_Comm comm)
```

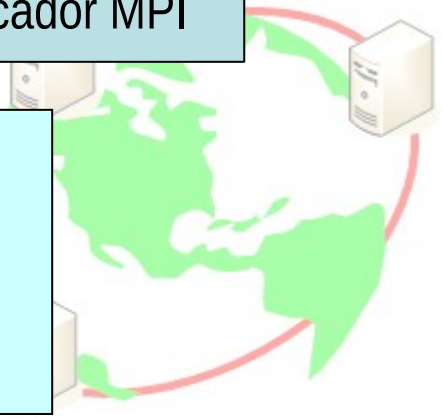
Processo destino

Rótulo da mensagem

Comunicador MPI

Exemplo:

```
int vetor[10]={0,1,2,3,4,5,6,7,8,9};
MPI_Send (vetor, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
```



HPCLab

Principais Primitivas MPI

- Recebimento de Mensagens

Endereço inicial da variável que receberá os dados

Número de elementos a serem recebidos

Tipo do dado

```
MPI_Recv (void *buf, int count, MPI_Datatype dtype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Processo fonte esperado

Rótulo esperado

Comunicador MPI

Processo fonte e rótulo efetivos

Exemplo:

```
int vetor[10];
MPI_Status status;   /*  status.MPI_SOURCE  e  status.MPI_TAG  */
MPI_Recv (vetor, 10, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                                              MPI_COMM_WORLD, &status);
```

Primitivas MPI

- Comunicação Coletiva

Endereço inicial dos dados a serem enviados ou recebidos

Número de elementos a serem enviados/recebidos

Tipo do dado

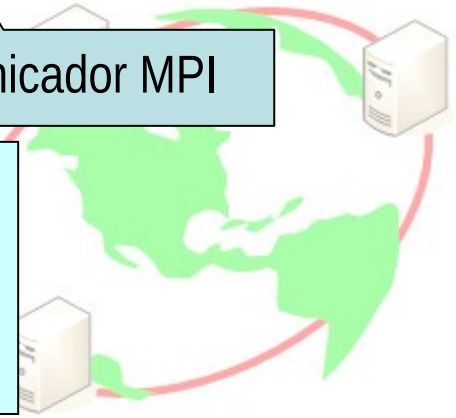
```
MPI_Bcast (void *buf, int count, MPI_Datatype dtype,
           int root, MPI_Comm comm)
```

Processo único que envia dados (os demais apenas recebem tais dados)

Comunicador MPI

Exemplo:

```
int vetor[50];
MPI_Bcast (vetor, 50, MPI_INT, 0, MPI_COMM_WORLD);
```



Primitivas MPI

- Comunicação Coletiva com Redução

Endereço inicial dos dados que serão enviados por cada processo

Endereço da variável que receberá os dados resultantes (processados)

Número de elementos a serem enviados/recebidos

Comunicador MPI

```
MPI_Reduce (void *sendbuf, void *recvbuf, int count,
MPI_Datatype dtype,MPI_Op op,int root,MPI_Comm comm)
```

Tipo do dado

Operação de processamento feita nos dados enviados, gerando os dados resultantes recebidos por "root"

Processo único que recebe os dados resultantes

Exemplo:

```
int total, parcial;
MPI_Reduce (&parcial, &total, 1, MPI_INT, MPI_SUM, 0,
                                          MPI_COMM_WORLD);
```

HPCLab

Primitivas MPI

Endereço inicial dos dados que serão enviados para cada processo

Número de elementos a serem enviados

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

Tipo do dado

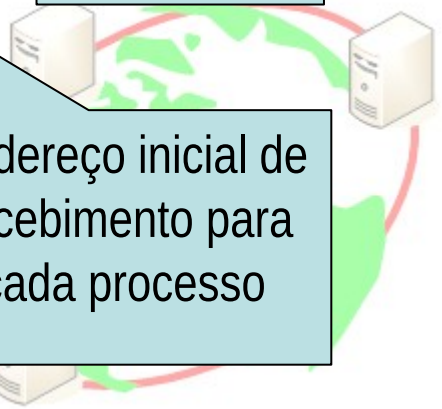
Tipo do dado

Processo único que recebe os dados resultantes

Comunicador MPI

Endereço inicial de recebimento para cada processo

Número de elementos a serem recebidos



HPCLab

Primitivas MPI

Endereço inicial dos dados que serão enviados

Número de elementos a serem enviados

Endereço inicial de recebimento para o processo root

```
MPI_Gather(void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

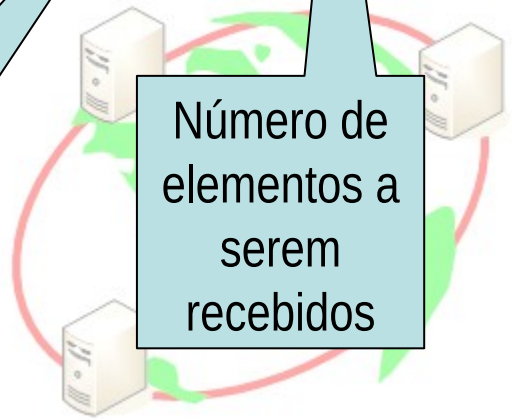
Tipo do dado

Tipo do dado

Processo único que recebe os dados resultantes

Comunicador MPI

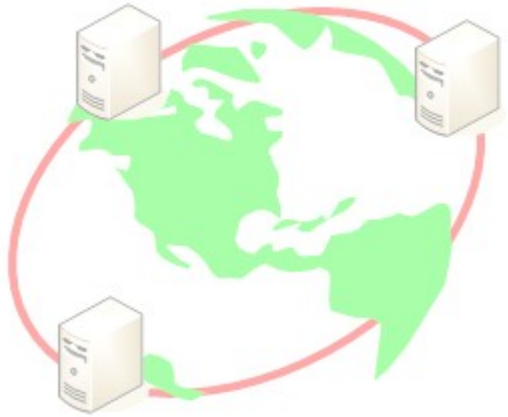
Número de elementos a serem recebidos



HPCLab

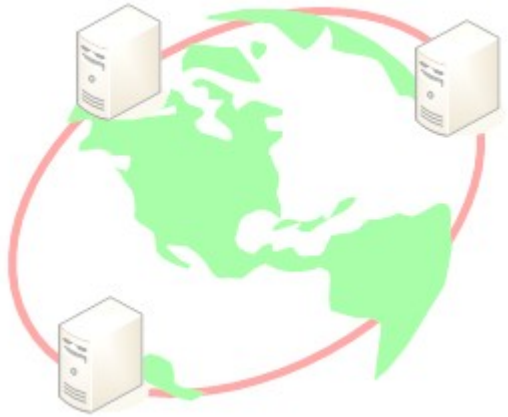
Operações de Redução MPI

- MPI_MAX (maximum)
- MPI_MIN (minimum)
- MPI_SUM (sum)
- MPI_PROD (product)
- MPI_LAND (logical and)
- MPI_LOR (logical or)
- MPI_LXOR (logical exclusive or)
- MPI_BAND (bitwise and)
- MPI_BOR (bitwise or)
- MPI_BXOR (bitwise exclusive or)



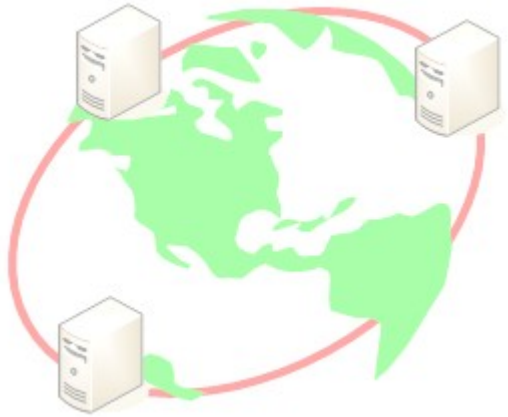
Exemplo de um programa MPI

- Os processos deverão determinar o maior elemento de um vetor
- Modelo mestre-escravo
  - Um processo distribui as tarefas (mestre)
  - Os demais (escravos) executam e retornam o resultado ao mestre
- 2 Estratégias
  - Comunicação ponto-a-ponto
  - Comunicação coletiva



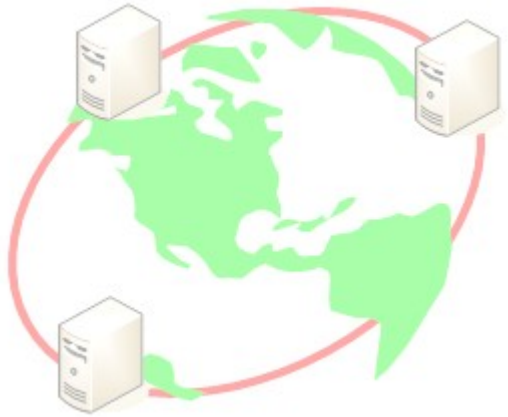
Rotina Principal

```
#include "mpi.h"
#include <stdio.h>

int numProcs, rank;
int mestre();
int escravo();

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank==0) mestre();
    else        escravo();
    MPI_Finalize();
}
```



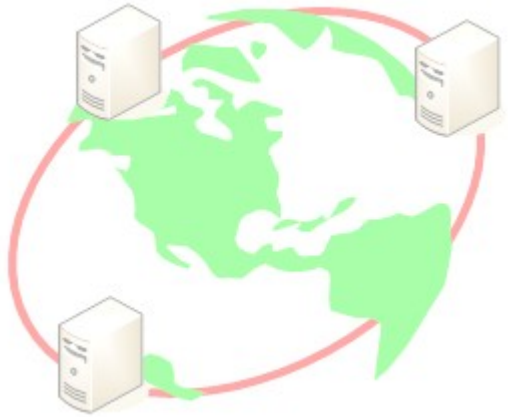
Estratégia 1 – Procedimento Mestre

```
int mestre() {
/* processo irá popular um conjunto de vetores de maneira aleatória e
   envia-los a seus escravos para processamento */
    MPI_Status status;
    int i, j, maior_parcial, maior, vetor[100];

    for(i=1; i<numProcs; i++) {
        for(j=0; j<100; j++)
            vetor[j]=numero_aleatorio();  // função qualquer pré-definida
        MPI_Send(vetor, 100, MPI_INT, i, 1, MPI_COMM_WORLD);
    }
    maior=0;
    for(i=1; i<numProcs; i++) {
        MPI_Recv(&maior_parcial, 1, MPI_INT, MPI_ANY_SOURCE,
                        MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        if(maior<maior_parcial)
            maior=maior_parcial;
    }
    printf("Maior entre %d elementos: %d\n", 100*(numProcs-1), maior);
}
```



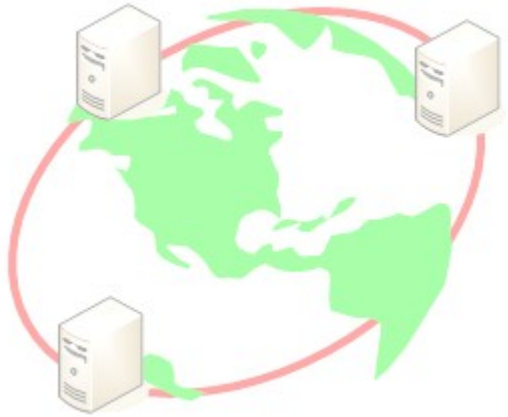
Estratégia 1 – Procedimento Escravo

```
int escravo() {
    /* processo irá receber parcialmente um vetor e
       determinar seu máximo valor local */
    MPI_Status status;
    int j, maior_parcial, vetor[100];

    MPI_Recv(vetor, 100, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    maior_parcial=0;
    for(j=0; j<100; j++) {
        if(maior_parcial<vetor[j])
            maior_parcial=vetor[j];
    }
    MPI_Send(&maior_parcial, 1, MPI_INT, 0,1,MPI_COMM_WORLD);
}
```



Estratégia 2 – Procedimento Mestre

```
int mestre() {
    MPI_Status status;
    int i, maior_parcial, maior, vetor[1000];

    for(i=0; i<1000; i++)
        vetor[i]=numero_aleatorio();  // função qualquer pré-definida
   /* envio o vetor como um todo */
    MPI_Bcast(vetor, 1000, MPI_INT, 0, MPI_COMM_WORLD);
    maior_parcial=0;
    for(i=0; i<1000/numProcs; i++) {
        if(maior_parcial<vetor[i])
            maior_parcial=vetor[i];
    }
    MPI_Reduce(&maior_parcial, &maior, 1, MPI_INT, MPI_MAX, 0,
                                                    MPI_COMM_WORLD);
    printf("Maior entre 1000 elementos: %d\n", maior);
}
```



Estratégia 2 – Procedimento Escravo

```
int escravo() {
    MPI_Status status;
    int i, inicio, fim, maior_parcial, maior, vetor[1000];
    // recebo todo o vetor e trato apenas a parte que sou responsável
    MPI_Bcast(vetor, 1000, MPI_INT, 0, MPI_COMM_WORLD);

    inicio=(1000/numProcs)*rank;
    fim=(1000/numProcs)*(rank+1);

    if(fim>1000) fim=1000;
    maior_parcial=0;
    for(i=inicio; i<fim; i++) {
        if(maior_parcial<vetor[i])
            maior_parcial=vetor[i];
    }
    MPI_Reduce(&maior_parcial, &maior, 1, MPI_INT, MPI_MAX, 0,
                                                    MPI_COMM_WORLD);
}
```